Empirical Evaluation of Stochastic Gradient Descent Variants for Training Convolutional Neural Networks on Image Classification Task

Eyasu Nigussie

March 2024

1 Background

Training a deep neural network involves minimizing a loss function, which is computed as the average of the errors between the model's predicted outputs and the true labels for each input sample across the entire training dataset or a representative subset (mini-batch) of the training data. To minimize the loss function during the training process, variants of the Stochastic Gradient Descent (SGD) algorithm could be employed. In this paper, I empirically compare the performance of four popular SGD variants – Momentum, Adagrad, RMSProp, and Adam – in terms of their convergence behavior when minimizing the loss function of a Convolutional Neural Network (CNN) trained for the task of classifying images of cats and dogs.

2 Overview of SGD Algorithms

2.1 Momentum

The Momentum algorithm is a variant of the Stochastic Gradient Descent (SGD) optimization algorithm used for training deep neural networks. It introduces a momentum term that accumulates gradients over time, allowing the optimizer to accelerate in the direction of the accumulated gradients instead of the steepest descent.

Let $f(w)$ denote the loss function, where w represent the weights of the model. In standard SGD, the weights are updated using the current gradient as follows:

$$w_{k+1} \leftarrow w_k - \eta \nabla f(w_k) \tag{1}$$

Where η is the learning rate, and $\nabla f(w_k)$ is the gradient of the loss function with respect to the weights at iteration k.

Momentum introduces a velocity vector, v, which accumulates the exponentially moving average of past gradients. The update rules for Momentum are:

$$v_{k+1} \leftarrow \beta v_k + \mathbf{g}_{k,k-1} \tag{2}$$

$$w_{k+1} \leftarrow w_k - \eta v_{k+1} \tag{3}$$

Where β is the scalar momentum parameter(typically set to 0.9), η is the learning rate, and $\mathbf{g}_{k,k-1}$ is the stochastic gradient of a random sample using the weights updated at $k-1$. When $\beta = 0$, the algorithm reduces to standard SGD.

One of the key advantages of the Momentum algorithm is its ability to escape shallow local minima. However, a limitation of the Momentum algorithm arises when dealing with tasks that involve features with varying frequencies. When using a fixed learning rate, the weights associated with frequently occurring features tend to converge faster than the weights associated with less frequently occurring features. This imbalance in convergence rates can lead to suboptimal performance.

2.2 Adaptive Gradient(Adagrad)

The Adagrad optimizer is a variant of the Stochastic Gradient Descent (SGD) algorithm that is designed to handle sparse data and features with varying frequencies. It adapts the learning rate for each parameter based on the squared sum of previous gradients. Some tasks, such as NLP, have features like words that occur less frequently than others. In such cases, the weights for common features converge faster than the weights of infrequent features. Adagrad assigns higher learning rates to infrequent features and makes the parameter update rely more on relevance than frequency of occurrence.[1] The learning rate is scaled by the aggregate of the squared previous gradients. It assigns a higher learning rate for features that occur less frequently.

The update rules for Adagrad are:

$$s_k(i) \leftarrow \sum_{l=1}^{k} g_l^2(i)$$

$$w_{k+1}(i) \leftarrow w_k(i) - \frac{\eta g_k(i)}{\sqrt{\epsilon + s_k}}$$

Where s_k is the accumulated squared gradient, η is an external learning rate, and ϵ is a small positive number to avoid divison by 0.

One of the limitations of Adagrad is its sensitivity to the selection of hyperparameters. This sensitivity means that the performance of Adagrad can be significantly influenced by the hyperparameters chosen. If the gradient is high initially, the learning rate be very low for the following iterations. This causes the learning to stall before arriving at optimal weights.

2.3 Root Mean Square Propagation(RMSProp)

Root Mean Square Propagation is a stochastic gradient descent optimization algorithm that addresses the problem of diminishing learning rates encountered in the AdaGrad algorithm.[5] AdaGrad accumulates the squared gradients over time, leading to a monotonically decreasing learning rate, which can cause premature convergence. RMSProp resolves this issue by employing an exponential moving average of the squared gradients, effectively giving more weight to recent gradients.

Let $f(\mathbf{w})$ denote the loss function, where $\mathbf{w}$ represents the model parameters. The update rule for RMSProp is as follows:

$$\mathbf{s}_k \leftarrow \gamma \mathbf{s}_{k-1} + (1-\gamma)\nabla_{\mathbf{w}} f(\mathbf{w}_k)^2 \tag{4}$$

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \frac{\eta}{\sqrt{\mathbf{s}_k + \epsilon}} \nabla_{\mathbf{w}} f(\mathbf{w}_k) \tag{5}$$

where:

- $\mathbf{s}_k$ is the exponential moving average of the squared gradients at time step k.
- γ is hyperparameter that controls the exponential decay rate of the moving average.
- $\nabla_{\mathbf{w}} f(\mathbf{w}_k)$ is the gradient of the loss function with respect to the parameters $\mathbf{w}$ at time step t.
- η is the learning rate.
- ϵ is a small constant added to avoid division by zero.

The key advantage of RMSProp over AdaGrad is that it adjusts the learning rate for each parameter based on recent gradients, rather than accumulating all past gradients.

2.4 Adaptive Moment Estimation(ADAM)

Adam (Adaptive Moment Estimation) is a stochastic gradient descent optimization algorithm that combines the advantages of two popular algorithms: Momentum and RMSProp. It computes adaptive learning rates for each parameter, utilizing estimates of both the first and second moments of the gradients. Adam is widely adopted for its effectiveness in various optimization problems and its ability to handle non-stationary objectives and sparse gradients.[2]

Let $f(\mathbf{w})$ denote the objective function, where $\mathbf{w}$ represents the model parameters. The Adam update rule is defined as follows:

$$\mathbf{g}_k = \nabla_{\mathbf{w}} f(\mathbf{w}_k) \tag{6}$$

$$\mathbf{m}_k \leftarrow \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\mathbf{g}_k \tag{7}$$

$$\mathbf{v}_k \leftarrow \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)\mathbf{g}_k^2 \tag{8}$$

$$\hat{\mathbf{m}}_k = \frac{\mathbf{m}_k}{1-\beta_1^k} \tag{9}$$

$$\hat{\mathbf{v}}_k = \frac{\mathbf{v}_k}{1-\beta_2^k} \tag{10}$$

$$\mathbf{w}_k \leftarrow \mathbf{w}_{k-1} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_k}+\epsilon}\hat{\mathbf{m}}_k \tag{11}$$

where:

- $\mathbf{g}_k$ is the gradient of the objective function with respect to the parameters $\mathbf{w}$ at time step k.
- $\mathbf{m}_k$ is the exponential moving average of the gradients (first moment).
- $\mathbf{v}_k$ is the exponential moving average of the squared gradients (second moment).
- β_1 and β_2 are the exponential decay rates for the first and second moment estimates, respectively. Typical values are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.
- $\hat{\mathbf{m}}_k$ and $\hat{\mathbf{v}}_k$ are the bias-corrected first and second moment estimates, respectively.
- η is the learning rate.
- ϵ is a small constant added to avoid division by zero.

Adam has become a popular choice due to its simplicity, computational efficiency, and strong empirical performance across a wide range of applications.[5]

3 Overview of Stepsizing Schemes

3.1 Learning Rate Decay

Learning rate decay is a technique used in training neural networks to improve convergence and generalization performance. It involves adjusting the learning rate, which controls the step size in the optimization process, during the training procedure. The learning rate is initially set to a relatively large value to facilitate rapid progress in the initial stages of training.[4] One common approach is the step decay, where the learning rate is reduced by a constant factor after a predefined number of epochs.

3.2 Cosine Annealing

Cosine Annealing is a learning rate scheduling technique that adjusts the learning rate during the training process of neural networks. Unlike step-based decay methods, Cosine Annealing follows a cyclical learning rate schedule, which has been shown to improve convergence and generalization performance.[3]

The key idea behind Cosine Annealing is to simulate a "warm restart" of the learning process by periodically resetting the learning rate to an increased value.

The learning rate update rule for Cosine Annealing is given by:

$$\eta_t = \eta_T + \frac{1}{2}(\eta_0 - \eta_T)(1+\cos(\pi\frac{t}{T})) \tag{12}$$

where:

- η_t is the learning rate at epoch t.
- η_0 is the initial learning rate.
- η_T is the minimum learning rate.
- T is the period (in epochs) after which the learning rate cycle restarts.

4 Empirical Comparison of Stochastic Gradient Descent Variants

To empirically compare the performance of Momentum, Adagrad, RMSProp and Adam, an image classification task was chosen, and a 17-layer Convolutional Neural Network (CNN) was trained using binary cross-entropy as the loss function. The dataset used was the Cats vs. Dogs dataset from Kaggle, consisting of 25,000 images, and the training was performed on a Google Colab V100 GPU.

4.1 Experimental Setup

The following optimizers were evaluated:

- Adam: Learning rate of 0.0001, default ϵ value, 10 epochs, 300 steps per epoch.
- RMSProp: Learning rate of 0.0001, default ϵ value, 10 epochs, 300 steps per epoch.
- Adagrad: Learning rate of 0.001, initial accumulator value of 0.1, default ϵ value, 10 epochs, 300 steps per epoch.
- Momentum: Learning rate of 0.001, momentum value of 0.9, 10 epochs, 300 steps per epoch.

4.2 Results and Analysis

The following table and image presents the training loss and accuracy for each optimizer over the 10 epochs:

Table 1: Training Performance of SGD Variants

Epoch	RMSProp		Adagrad		Momentum		Adam	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
1	0.698799	0.566667	0.689557	0.533750	0.687222	0.540000	0.667482	0.586458
2	0.642505	0.620521	0.681993	0.556667	0.674791	0.570833	0.617813	0.646667
3	0.614982	0.660208	0.673079	0.575417	0.663259	0.590937	0.605455	0.668958
4	0.598351	0.673437	0.666776	0.591875	0.650619	0.611042	0.581555	0.687708
5	0.583674	0.688958	0.659703	0.602292	0.642300	0.623229	0.572500	0.698021
6	0.572713	0.694687	0.654176	0.607500	0.629468	0.639583	0.552156	0.712500
7	0.563867	0.704167	0.650189	0.613646	0.622863	0.651771	0.541882	0.716042
8	0.555474	0.712396	0.643196	0.625104	0.617409	0.654896	0.543596	0.724063
9	0.545283	0.716771	0.635417	0.631771	0.602997	0.667188	0.519906	0.739896
10	0.539768	0.717708	0.638652	0.627292	0.597519	0.674792	0.521970	0.735937

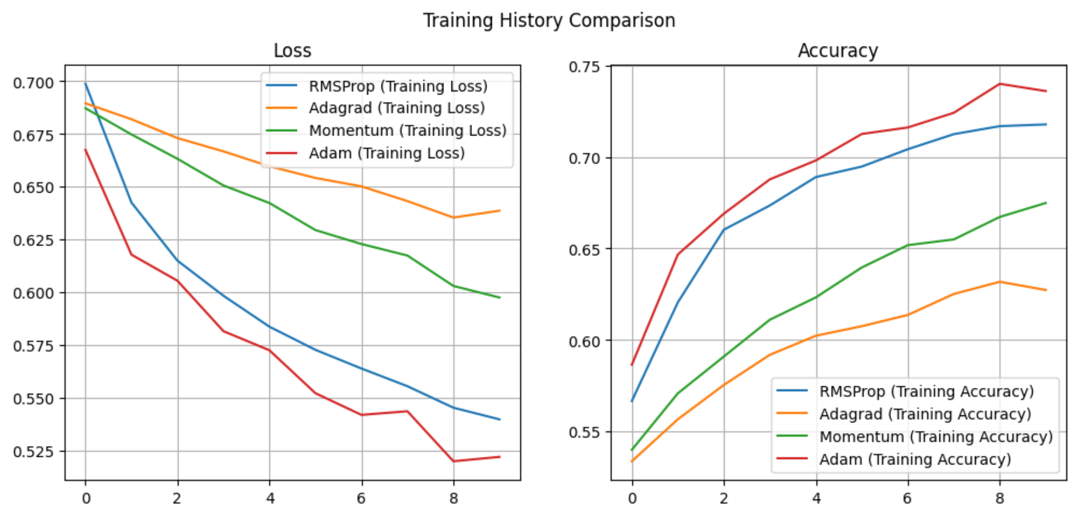

Figure 1: Training loss and accuracy curves for each optimizer

As observed from the table and Figure 1, Adam performed exceptionally well, starting with a low loss value in the first epoch and decreasing it further in the following epochs. On the other hand, Adagrad exhibited relatively the lowest performance. RMSProp also demonstrated strong performance, outperforming Adagrad and Momentum. Adagrad struggled in this task due to its tendency to accumulate squared gradients over time and updates became very small with each iteration.

5 Conclusion

The empirical comparison highlights the performance differences among the SGD variants for the image classification task. Adam emerged as the top-performing, followed by RMSProp, Momentum, and Adagrad.

Training Code

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense,
    Concatenate
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

def create_and_train_model(train_dir, validation_dir, model_name=None,
    learning_rate=None, momentum=None, initial_accumulator=None):

    train_datagen = ImageDataGenerator(
        rescale=1/255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.1,
        horizontal_flip=True,
        fill_mode='nearest'
    )

    validation_datagen = ImageDataGenerator(rescale=1/255)

    train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(256, 256),
        color_mode='rgb',
        class_mode='binary',
        batch_size=32
    )

    validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(256, 256),
        color_mode='rgb',
        class_mode='binary',
        batch_size=32
    )


    input_shape = (256, 256, 3)
    inputs = Input(shape=input_shape)

    conv1 = Conv2D(32, (3,3), activation='relu', padding='same')(inputs)
    conv2 = Conv2D(32, (5,5), activation='relu', padding='same')(inputs)
    conv3 = Conv2D(32, (7,7), activation='relu', padding='same')(inputs)

    concatenated1 = Concatenate()([conv1, conv2, conv3])
    pooling1 = MaxPooling2D(pool_size=(2,2))(concatenated1)

    conv4 = Conv2D(64, (3,3), activation='relu', padding='same')(pooling1)
    conv5 = Conv2D(64, (5,5), activation='relu', padding='same')(pooling1)
    conv6 = Conv2D(64, (7,7), activation='relu', padding='same')(pooling1)

    concatenated2 = Concatenate()([conv4, conv5, conv6])
    pooling2 = MaxPooling2D(pool_size=(2,2))(concatenated2)
    flatten = Flatten()(pooling2)

    dense1 = Dense(256, activation='relu')(flatten)
    dense2 = Dense(128, activation='relu')(dense1)
    dense3 = Dense(64, activation='relu')(dense2)
    outputs = Dense(1, activation='sigmoid')(dense3)

    if model_name == 'Adam':
        optimizer = Adam(learning_rate=learning_rate)
    elif model_name == 'Momentum':
        optimizer = optimizers.SGD(learning_rate=learning_rate, momentum=momentum)
    elif model_name == 'RMSProp':
        optimizer = optimizers.RMSprop(learning_rate=learning_rate)
    else:
        optimizer =
            optimizers.Adagrad(learning_rate=learning_rate,initial_accumlator_value=0.1)

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(
        optimizer=optimizer,
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    train_history = model.fit(
        train_generator,
        steps_per_epoch=300,
        epochs=10,
        batch_size=32,
        validation_data=validation_generator,
        validation_steps=100
    )

    return model, train_history
```

Listing 1: Python code for creating and training a neural network model

References

[1] *AdaGrad - Cornell University Computational Optimization Open Textbook - Optimization Wiki.* URL: `https://optimization.cbe.cornell.edu/index.php?title=AdaGrad`.

[2] Jason Brownlee. *Gentle introduction to the adam optimization algorithm for deep learning.* 2021. URL: `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`.

[3] Ilya Loshchilov and Frank Hutter. *SGDR: Stochastic Gradient Descent with Warm Restarts.* 2017. arXiv: `1608.03983 [cs.LG]`.

[4] Kaichao You et al. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. arXiv: `1908.01878 [cs.LG]`.

[5] Aston Zhang et al. *Dive into Deep Learning.* `https://D2L.ai`. Cambridge University Press, 2023.